

ERDC

Major Shared Resource Center

MSRC



ERDC MSRC/PET TR/00-02

MDARUN - A Package of Software For Creating Multidisciplinary Applications with MPI

by

Richard Weed

**Work funded by the DoD High Performance Computing
Modernization Program CEWES
Major Shared Resource Center through**

Programming Environment and Training (PET)

Supported by Contract Number: DAHC94-96-C0002
Nichols Research Corporation

Views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of Defense Position, policy, or decision unless so designated by other official documentation.

MDARUN - A PACKAGE OF SOFTWARE FOR CREATING MULTIDISCIPLINARY APPLICATIONS WITH MPI

Richard Weed *
Mississippi State University

November 2, 1999

1 INTRODUCTION

The purpose of this report is to document a suite of FORTRAN and C routines and UNIX C (csh) shell scripts that have been developed to reduce the effort required to implement multidisciplinary applications using the Message Passing Interface (MPI) library [1]. A procedure for implementing multidisciplinary applications with MPI and a brief introduction to the software described in this report are presented in an accompanying report [2]. As discussed in [2], the development of the software was prompted by a need for a suitable replacement for the MPIRUN software package [3] to support ongoing U.S. Army Engineer Research and Development Center (ERDC) research [4]. In addition, there is a clear need for a set of easily maintained routines that simplify the creation of multidisciplinary applications by providing a standard procedure for defining the data required to set up a multidisciplinary application and calling the required MPI functions.

The following sections of this report will describe in detail the FORTRAN 77 and FORTRAN 90 subroutines and C functions that make up a package of software named MDARUN. A general description of each routine will be given followed by discussion of the argument list data, I/O data, COMMON block data, FORTRAN 90 module data, and C external variables required for each routine. Guidelines for using the routines and examples are given for both the FORTRAN and C versions of the software.

2 OVERVIEW OF THE MDARUN SOFTWARE

The MDARUN software package consists of one primary routine and four supporting utility routines. The primary routine is *MDA_Init*, which is responsible for creating the MDARUN arrays and other data needed to create a group communicator for each application group defined by the user. The user supplies *MDA_Init* with the following four data items: the number of application types, the number of application groups to be used in a run, the number of processes in each application group, and an index that is used to identify a group application type. This information can be either hard coded into a user's code prior to a call to *MDA_Init* or read from a user-defined data file inside the *MDA_Init* routine. All applications in a multidisciplinary application must call *MDA_Init* at approximately the same place in their logic streams.

The four supporting utilities are *MDA_Intergroup_comm*, which returns a group intercommunicator for a given group number, *MDA_Sameapp_Comm* which returns an intracommunicator that contains all the applications started with the same application type, *MDA_Mergeproc_comm* which creates an intracommunicator based on a list of processor ranks that can span different (or the same) groups, and *MDA_Create_mergelist*, which is used to create merge lists for *MDA_Mergeproc_comm*. These routines provide wrappers around

* PET CSM Onsite Lead, ERDC MSRC, 1165 Porters Chapel Road, Vicksburg, MS 39180. E-mail: rweed@wes.hpc.mil

appropriate calls to MPI routines with simple argument list input and output. All of the MDARUN routines are designed around the group concept discussed in [2]. Therefore, the user supplies group-oriented data (group numbers, local ranks in groups, etc.) instead of global ranks in the global MPI communicator, `MPLCOMM_WORLD`.

In addition to the core MDARUN routines, the package supplies support routines for creating command files for the SGI/CRAY Origin2000 `mpirun` and the IBM SP `poe` and `pbspoe` commands that are used to start MPI jobs on those systems.

3 DETAILED DESCRIPTIONS OF THE MDARUN FUNCTIONS AND SUBROUTINES

3.1 MDA_Init

3.1.1 MDA_Init Description

`MDA_Init` is responsible for defining the MDARUN data and creating the `MDARUN_GROUP_COMM` communicator. `MDA_Init` takes two arguments in its argument list (see the function/subroutine prototype and interface descriptions below). The first argument, `no_read`, controls whether `MDA_Init` will try to read its required initialization data from a data file or assumes the data are defined by the user prior to the call to `MDA_Init`. Setting `no_read` to zero tells `MDA_Init` to read the required data from a file. For any nonzero value of `no_read`, `MDA_Init` assumes the user has defined the MDARUN data before `MDA_Init` is called.

The second argument list variable is `md_unit` which is an integer logical I/O unit number in FORTRAN or a FILE pointer in C. Both the FORTRAN and C versions of the code will check to see if a file associated with `md_unit` has been opened prior to the call to `MDA_Init`. If `no_read` is zero and no file is associated with `md_unit` at the time of the call to `MDA_Init`, the subroutine will default to reading the MDARUN input data from a file with the name “`mdainit.dat`” in the local directory.

3.1.2 Data Required by MDA_Init

In `MDA_Init`, the process with global rank zero is the only process that is allowed to read the input data file. The input data, once defined, is broadcast to all the other processes by a call to `MPI_Bcast`. The following data variables and arrays must be defined either by input or hard coded into the user’s application:

- | | |
|-------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>num_app_types</code> | - The number of different application types
(Integer variable) |
| <code>num_app_groups</code> | - The number of application groups defined
by the user (Integer variable) |
| <code>nprocs_per_group</code> | - The number of processes in each group
(Integer array dimensioned from 0 to <code>max_groups-1</code>
where <code>max_groups</code> is defined by the user) |
| <code>group_app_type</code> | - An index (i.e. 1, 2, etc.) that is used
to identify each different application type.
(Integer array dimensioned from 0 to <code>max_groups-1</code>
where <code>max_groups</code> is defined by the user) |

The FORTRAN 90 and C versions of `MDA_Init` use dynamic memory allocation for all arrays. Array sizes are based on the total number of processes detected in `MPLCOMM_WORLD`. The user is responsible for performing the appropriate ALLOCATEs or `mallocs` if the data described above are to be hard coded into the application.

3.1.3 MDA_Init Function/Subroutine Prototype and Interface

The *MDA_Init* function has the following FORTRAN interface and C prototype:

FORTRAN:

```
Subroutine MDA_Init(no_read, md_unit)
    Integer no_read, md_unit
```

C:

```
void MDA_Init(int no_read, FILE *md_unit);
```

3.1.4 Data Exported by MDA_Init

Each version of the MDARUN software (FORTRAN 77, FORTRAN 90, and C) exports the same data but by different mechanisms (i.e., COMMON blocks, FORTRAN 90 modules, or external variables in C). The definitions and names of the data exported by *MDA_Init* are as follows:

mdarun_group_id	- The group number of each process (Integer variable)
mdarun_num_groups	- The number of application groups in the current job (Same as num_app_groups) (Integer variable)
MDARUN_GROUP_COMM	- The group intracommunicator created by <i>MDA_Init</i> . It should be all caps in C. (Integer variable in FORTRAN, defined type MPI_Comm in C)
mdarun_group_leaders	- Integer array containing global rank in MPI_COMM_WORLD of the lead process (local rank = zero) in each group (Dimension is 0 to max_procs-1 where max_procs is either the total number of processes started for the job or a maximum value set by the FORTRAN 77 PARAMETER statement)
nprocs_per_group	- Defined previously
group_app_type	- Defined previously
global_to_group_map	- An integer array containing the group number of every process in MPI_COMM_WORLD. (Dimension is 0 to max_procs-1 where max_procs is either the total number of processes started for the job or a maximum value set by the FORTRAN 77 PARAMETER statement)

In addition, the following MPIRUN variables are set to their MDARUN equivalents:

```
MPIRUN_APP_ID      -> mdarun_group_id
MPIRUN_NUM_APPS    -> mdarun_num_groups
MPIRUN_APP_COMM    -> MDARUN_GROUP_COMM (Duplicated by MPI_Comm_dup)
MPIRUN_APP_LEADERS -> mdarun_group_leaders
```

C programmers should use the capitalization shown above for all variables.

3.1.5 FORTRAN 77 INCLUDE File

For the FORTRAN 77 version, all the data are passed in three different named COMMON blocks: MDARUN, MDADAT, and MPIRUN. The MPIRUN COMMON contains the same data items as the COMMON block of the same name in the old MPIRUN package [3]. The array sizes must be set by PARAMETER statements. An INCLUDE file, mdarunf.h, is provided as part of the MDARUN package that contains the

COMMON blocks and appropriate PARAMETER statements. The contents of mdarunf.h are shown in Appendix A. The user should INCLUDE mdarunf.h in any routines that will use the data generated by *MDA_Init*.

3.1.6 FORTRAN 90 Modules

For the FORTRAN 90 version of MDARUN, the MDARUN data are exported using two FORTRAN 90 modules, “mdarun_data” and “mpirun_data”. The user is responsible for inserting USE mdarun_data and/or USE mpirun_data at the appropriate places in their code. The mdarun.data and the mpirun.data modules are shown in Appendix A.

3.1.7 C Header File

A header file, “mdarun.h”, that contains the appropriate type definitions and function prototypes for the MDARUN software is provided for C applications. This file must be included after the standard MPI header file, “mpi.h”, because mdarun.h uses MPI-defined types that are set in mpi.h. The file mdarun.h must be included outside of the user’s main function. The header file is shown in Appendix A.

3.1.8 Using MDA_Init

The *MDA_Init* function or subroutines should always be called after the user’s call to *MPI_Init*. The user has three options for entering the data required by *MDA_Init*. The first option is to set the *no_read* variable to some nonzero number and manually set the appropriate variables and array values prior to the call to *MDA_Init*. For this option, the user is responsible for allocating space for the appropriate MDARUN arrays. The second option is to set *no_read* to zero and open a user-defined data file prior to the call to *MDA_Init*. The user must pass the logical unit number in FORTRAN or a pointer of type FILE in C as the *md_unit* argument. *MDA_Init* will attempt to read the required data from the user-defined file. The third option is to let *MDA_Init* read the required data from its default file, mdainit.dat. When *no_read* is set to zero, the code will check to see if a data file exists and abort if it does not detect either a user-defined file or a file named “mdainit.dat” in the local directory. Only one call to *MDA_Init* is required for any application. However, all applications must call *MDA_Init* in approximately the same place in the logic stream irrespective of the application types or languages to ensure that the group communicators and other MDRARUN data are defined consistently across all processes.

The following code fragments illustrate how *MDA_Init* is called in FORTRAN and C code:

FORTRAN:

```
no_read=0
md_unit=7
Open(md_unit, file="my_data_file", STATUS="Unknown")
Call MDA_Init(no_read, md_unit)
```

C:

```
no_read=0;
md_unit=fopen("my_data_file","r");
MDA_Init(no_read, md_unit);
```

3.1.9 MDA_Init Data File Format

The data file required by *MDA_Init* when the *no_read* variable is set to zero has the following format:

```
num_app_types  num_app_groups
2 4
nprocs/group  group_app_type
2 1
2 2
3 1
3 2
```

The non-numeric data are two character header strings (40 characters max). The first line of numeric data are the values for *num_app_types* and *num_app_groups*. In this example there are two unique application types and four different groups. The remaining numeric data define the number of processes per group (*nprocs_per_group*) and an index defining the application type (*group_app_type*). There is one line of input for each of the four groups. All numeric data should be separated by at least one blank character. The two character header strings are required but can be blank if the user desires.

3.2 MDA_Intergroup_comm

3.2.1 MDA_Intergroup_comm Description

The routine, *MDA_Intergroup_comm*, acts as a wrapper routine around a call to the MPI function, *MPI_Intercomm_create*. The purpose of *MDA_Intergroup_comm* is to simplify the creation of group intercommunicators. As shown below in the function and subroutine prototypes, *MDA_Intergroup_comm* requires only three arguments: the group number of the remote group to which you wish to build an intercommunicator, a safe tag for communications, and an intercommunicator (an integer in FORTRAN or a MPI defined type in C). The C version returns a pointer so the intercommunicator name should be prefixed with the C address operator (i.e., &MY_COMM instead of MY_COMM) unless the communicator was also declared to be a pointer (i.e., MPI_Comm *MY_COMM instead of MPI_Comm MY_COMM). All other data used by *MDA_Intergroup_comm* is generated by *MDA_Init* and passed internally by COMMON blocks, FORTRAN 90 modules, or C external variables.

3.2.2 MDA_Intergroup_comm Function/Subroutine Prototype and Interface

The FORTRAN interface and C prototype for the *MDA_Intergroup_comm* routine are as follows:

FORTRAN:

```
Subroutine MDA_Intergroup_comm(group_num, comm_tag, group_to_group_com)
  Integer group_num, comm_tag, group_to_group_com
```

C:

```
void MDA_Intergroup_comm(int group_num, int comm_tag,
  MPI_Comm *group_to_group_com);
```

3.2.3 Using MDA_Intergroup_comm

The advantage of using *MDA_Intergroup_comm* over a standard call to *MPI_Intercomm_create* is that the user does not have to keep up with the ranks of the group leaders in the remote and local groups to create an intercommunicator. The user creates the intercommunicator by specifying the MDARUN group number of the remote group. The following FORTRAN and C code fragments illustrate the use of

MDA_Intergroup_comm to create an array of intercommunicators for all the groups defined for a given application. *MDA_Intergroup_comm* will return the group communicator, MDARUN_GROUP_COMM, whenever the specified group number is the same as the group number of the calling process.

FORTRAN:

```
comm_tag=0
Do n=0,mdarun_num_groups-1
  Call MDA_Intergroup_comm(n,comm_tag, INTERCOMM(n))
End Do
```

C:

```
comm_tag=0;
for( n=0;n<=mdarun_num_groups-1);n++ )
  MDA_Intergroup_comm(n,comm_tag, &INTERCOMM[n]);
```

3.3 MDA_Sameapp_comm

3.3.1 MDA_Sameapp_comm Description

The purpose of *MDA_Sameapp_comm* is to form a new intracommunicator that contains all application groups of the same type. The only data returned by *MDA_Sameapp_comm* is the new communicator which is passed as an argument. *MDA_Sameapp_comm* uses the *group_app_type* array to determine the application type of the current processor. A call to *MPI_Comm_split* is made with the application type set as the “color” variable. A typical use of *MDA_Sameapp_comm* would be to merge smaller groups of applications of the same type into a new, larger group with a single intracommunicator.

3.3.2 MDA_Sameapp_comm Function/Subroutine Prototype and Interface

The FORTRAN and C versions of *MDA_Sameapp_comm* have the following interface and prototype:

FORTRAN:

```
Subroutine MDA_Sameapp_comm(same_app_comm)
  Integer same_app_comm
```

C:

```
void MDA_Sameapp_comm(MPI_Comm *same_app_comm);
```

3.3.3 Using MDA_Sameapp_comm

The creation of the merged communicator requires only a single call to *MDA_Sameapp_comm*. As with the *MDA_Intergroup_comm* routine, C programmers should return the address of the new communicator unless the variable is defined as a pointer.

FORTRAN:

```
Call MDA_Sameapp_comm(SAMETYPE_COMM)
```

C:

```
MDA_Sameapp_comm(&SAMETYPE_COMM); /* SAMETYPE_COMM is typed MPI_Comm */
```

or

```
MDA_Sameapp_comm(SAMETYPE_COMM); /* SAMETYPE_COMM is typed MPI_Comm* */
```

3.4 MDA_Mergeproc_comm and MDA_Create_mergelist

3.4.1 MDA_Mergeproc_comm and MDA_Create_mergelist Description

MDA_Mergeproc_comm will create a new intracommunicator that is constructed from a user-defined list of global ranks in MPI_COMM_WORLD. The current process should always be a member of the new communicator. The ranks in the merge list can span other groups or the group of the current process. The purpose of *MDA_Mergeproc_comm* is to allow the user to merge individual processes in different groups into a new unified intracommunicator. The user defines the following data via the routine argument list: a merge list that contains the global ranks in MPI_COMM_WORLD of the processes to be merged, the size of the merge list, and a user-defined merge_number that uniquely identifies each call to *MDA_Mergeproc_comm*. The routine returns the new intracommunicator as an argument.

Another utility routine, *MDA_Create_mergelist*, is provided to assist in the creation of the merge list. As with the other routines in MDARUN, *MDA_Create_mergelist* works with group data and not global ranks. *MDA_Create_mergelist* takes in arrays that define a list of group numbers for the individual processes that the user wishes to merge as input, a second array containing the corresponding local ranks in each group of the desired processes, and a list size. The routine returns a third integer array, the merge list. The user is responsible for allocating memory space for these arrays. In the C version of *MDA_Create_mergelist*, the arrays are passed as array pointers.

3.4.2 MDA_Mergeproc_comm and MDA_Create_mergelist Function/Subroutine Prototypes and Interfaces

The FORTRAN interfaces and C prototypes for *MDA_Mergeproc_comm* and *MDA_Create_mergelist* are:

FORTRAN:

```
Subroutine MDA_Mergeproc_comm(merge_list, list_size, merge_number, &
                             merged_proc_comm)
  Integer mergelist(0:list_size-1), list_size, merge_number, &
  merged_app_comm

Subroutine MDA_Create_mergelist(group_list, local_rank_list, &
                               list_size, merge_list)
  Integer group_list(0:list_size-1), local_rank_list(0:list_size-1)
  Integer list_size, merge_list(0:list_size-1)
```

C:

```
void MDA_Mergeproc_comm(int *merge_list, int list_size, int merge_number,
                        MPI_Comm *merged_proc_comm);

void MDA_Create_mergelist(int *group_list, int *local_rank_list, int list_size,
                         int *merge_list);
```

3.4.3 Using MDA_Mergeproc_comm and MDA_Create_mergelist

MDA_Mergeproc_comm and *MDA_Create_mergelist* are designed to work together. The user is responsible for providing the appropriate arrays and/or array pointers. The following FORTRAN and C code fragments illustrate how the two routines would be used to create a communicator that contains processes of the same local rank as the current process from different groups.

FORTRAN:

```
Do n=0, num_merges-1
    merge_num=n
    Do nn=0, mdarun_num_groups-1
        group_list(nn) = nn
        local_rank_list(nn) = n
    End Do
    Call MDA_Create_mergelist(group_list, local_rank_list, mdarun_num_groups, &
                             merge_list(0))
    Call MDA_Mergeproc_comm(merge_list(0), mdarun_num_groups, merge_num, &
                           tmp_comm)
    If (my_local_rank .eq. n) Then
!
! Use MPI_Comm_dup to create final version of communicator
! This is the safe way to do it in C. Not required for FORTRAN
! but should do it anyway if you want to connect a FORTRAN code to
! a C code. This insures that the communicator will have a consistent
! value
!
        Call MPI_Comm_dup(tmp_comm, MY_MERGED_COMM,ierr)
    EndIf
EndDo
```

C:

```
for(n=0;n<=num_merges-1;n++)
{
    merge_num=n;
    for{nn=0,nn<=mdarun_num_groups-1;nn++)
    {
        group_list[nn] = nn;
        local_rank_list[nn] = n;
    }
    MDA_Create_mergelist(group_list,local_rank_list,mdarun_num_groups,
                         merge_list);
    MDA_Mergeproc_comm(merge_list, mdarun_num_groups, merge_num,
                       &tmp_comm);
/*
!
! Use MPI_Comm_dup to create final version of communicator
! This is the safe way to do it in C. Not required for FORTRAN
! but should do it anyway if you want to connect a FORTRAN code to
! a C code. This insures that the communicator will have a consistent
! value
```

```

!
*/
if (my_local_rank==n)
    MPI_Comm_dup(tmp_comm, MY_MERGED_COMM);
}

```

4 STARTING MULTIDISCIPLINARY APPLICATIONS ON ERDC MSRC PARALLEL SYSTEMS

The information in the following sections describes the procedures for starting multidisciplinary jobs on ERDC MSRC SGI/CRAY Origin2000 and IBM SP parallel systems. Unfortunately, the loader commands on the ERDC MSRC CRAY T3E system do not allow more than one application to be started at a time from the same command. Whether this is particular to the ERDC MSRC T3E system or a generic feature of the CRAY T3E Message Passing Toolkit implementation of the *mpirun* loader command is being investigated.

4.1 STARTING MULTIDISCIPLINARY APPLICATIONS ON THE SGI/CRAY ORIGIN2000

The loader command for all MPI runs on the Origin2000 is *mpirun* (not the same *mpirun* as given in [3]). The following *mpirun* command starts two instances of a.out and two instances of b.out:

```
mpirun -np 2 a.out : -np 2 b.out
```

This command implies that there are two groups in this application, an a.out group of two processes and a b.out group of two processes. For applications with a large number of groups, it is more convenient to put the *mpirun* commands in a file and then use the -f(file) option on the *mpirun* command line instead of specifying the number of processes and executable file names on the command line. Placing the following lines in a file named cmdfile.sgi and typing *mpirun -f cmdfile.sgi* has the same effect as the previous *mpirun* command:

```
2 a.out :
2 b.out
```

The -np switch is not required to specify the number of processes when inputting the *mpirun* commands from a file.

4.2 STARTING MULTIDISCIPLINARY APPLICATIONS ON THE IBM SP SYSTEMS

All MPI jobs on the ERDC MSRC IBM SP systems (osprey and pandion) are started with the PBS (Portable Batch System) version of the IBM *poe* command, *pbspoe*. In normal usage, only one executable is specified on the *pbspoe* command. This is consistent with the default programming model assumed by PBS for most users, SPMD (Single Program Multiple Data). The type of programming model to be used by *pbspoe* is one of the options that can be specified on the *pbspoe* command line. For multidisciplinary jobs, the programming model must be changed to MPMD (Multiple Program Multiple Data). As with the SGI *mpirun* command, it is more convenient to enter the executable file names from an input file instead of specifying them on the command line. The *pbspoe* command for starting two instances of a.out and two instances of b.out would be written as:

```
pbspoe -procs 4 -pgmmodel mpmd -cmdfile ./cmdfile_ibm
```

where the file cmdfile_ibm would contain the following lines:

```
a.out  
a.out  
b.out  
b.out
```

Note that there is one line in cmdfile_ibm for each instance of a.out and b.out. The total number of lines should equal the total number of processors used for the job.

4.3 MDARUN UTILITIES FOR CREATING SGI AND IBM LOADER COMMAND FILES

Two small FORTRAN programs (make_sgi_run and make_ibm_run) have been written to assist users in creating the *mpirun* and *pbspoe* command files described in the two preceding sections. These programs read an input file that has the same format on both the SGI and IBM systems. After compiling the appropriate program, the user need only type:

```
make_sgi_run < my_data_file  
or  
make_ibm_run < my_data_file
```

The input file (my_data_file in the current example) would contain entries with the following format:

```
# Comments begin with #  
2 /u/my_dir/a.out  
# Comments are not required  
2 /u/my_dir/b.out  
2 /u/my_dir/c.out
```

The data file specifies that two instances each of a.out, b.out, and c.out will make up the final *mpirun* or *pbspoe* command file input. The programs will create output files named cmdfile_sgi or cmdfile_ibm.

In addition to the FORTRAN programs, two simple Unix C shell (csh) scripts, run_sgi_mda and run_ibm_mda, have been written that will run the make_sgi_run or make_ibm_run programs and then execute the appropriate *mpirun* or *pbspoe* command.

5 INCORPORATING THE MDARUN SOFTWARE IN USER APPLICATIONS

The routines in the MDARUN package are relatively small and designed to be added to a user's existing code base. Users are free to build their own libraries from the software if they prefer not to include the MDARUN source code into their applications. All versions of the software have been successfully compiled and tested on the ERDC MSRC SGI/CRAY Origin2000 and IBM SP systems. In addition, it is being used in the ongoing ERDC research project described in [4]

6 OBTAINING THE MDARUN SOFTWARE

Current plans are for the MDARUN software to be made available for public release via the ERDC MSRC web pages and the *PETtools* directory that is available on all of the ERDC MSRC machines. ERDC MSRC users wishing to obtain access to the MDARUN software should contact the ERDC MSRC Customer Assistance Center (info-hpc@wes.hpc.mil or (800) 500-4722).

7 SOME TYPICAL USES OF THE MDARUN SOFTWARE

The MDARUN software's primary use is to create an intercommunicator between two different programs in a multidisciplinary application. However, the group structure of the software allows more than one type of application to be built. For instance, multiple groups of the same application can be started with different processors in each group running different input data. This allows a user to use a parallel system to run multiple test cases for the same application concurrently in the same batch job or run submittal. Users who run on batch systems that allow only one job per user to be run at a time can utilize this feature to run multiple cases without having to submit multiple jobs that will wait in the batch system input queue.

The *MDA_Sameapp_comm* and *MDA_Mergeproc_comm* routines can be used to group the same or different applications together into new communicators. Therefore, these functions provide users with great flexibility in how they merge individual groups or processes together. For example, a user might wish to assign an instance of some data reduction or visualization application to each instance of their standard analysis code. The user can use the MDARUN utilities to build an intracomunicator that will contain one instance each of the analysis and data reduction or visualization code. The user can then assign a computational grid or grid partition to each of these new subgroups. Each instance of the analysis code now has a dedicated data reduction or visualization application that runs concurrently with the analysis code.

8 The MDARUN TEST CODE - *Test_mda*

FORTRAN 90 and C versions of a simple test program for the MDARUN software are given in Appendix B. The program *Test_mda* calls a unified setup routine, *Setup_mda*, that in turn calls the MDARUN software routines to create appropriate communicators. The *Setup_mda* routine can be used as a template for developers who wish to write a similar routine for their applications. *Test_mda* uses the communicators created by the call to *Setup_mda* to perform the following MPI communication tasks. First, the MDARUN_GROUP_COMM communicator is used by members of the same group to exchange one thousand double precision words of data between members of the group. Next, the SAMETYPE_COMM communicator is used in a *MPI_Allreduce* call to form a list of the process global ranks. The group intercommunicators are then used by the group leaders to send one thousand double precision words to each process in the remote groups and then receive the same information from the remote processes. Finally, the MY_MERGED_COMM communicator is used in two *MPI_Allreduce* operations to send the global rank and application type of each process to all the other processes in the communicator.

9 CONCLUDING REMARKS

This report has presented a detailed description of the subroutines and functions contained in the MDARUN software package. The software was designed to reduce the effort required by ERDC MSRC users to create multidisciplinary applications. In addition, the software was designed as a functional replacement for the MPIRUN software package [3].

References

- [1] Snir, M., et al., *MPI - The Complete Reference*, MIT Press, 1996.
- [2] Weed, R., "Building Multidisciplinary Applications With MPI," ERDC MSRC PET TR-00-01, August, 1999.
- [3] Fineberg, S., "Implementing Multi-disciplinary and Multi-zonal Applications Using MPI," Report No. NAS-95-003, NASA Ames Research Center, January, 1995.
- [4] Bernard, R., "Structured-Unstructured Modeling," CHSSI EQM NO. 1,
<http://www.hpcmo.hpc.mil/HTdoc/CTAs/index.html>, 1999.

APPENDIX A. MDARUN COMMON BLOCKS, F90 MODULES and C EXTERNAL VARIABLES

A.1 The MDARUN FORTRAN 77 INCLUDE File - mdarunf.h

```

C
C      MDARUN/MPIRUN Common blocks and parameters
C
C Definitions
C
C Parameter Data
C      max_groups, max_procs : Maximum number of groups
C                                an processes the user
C                                can use. Best to set these
C                                equal and as big as the
C                                largest number of processes
C                                the user expects to need
C      max_zones,           : Used to define size of MPIRUN_LEADERS
C                                array. Set = to max_groups
C      maxbuf               : maximum size of the buffer array
C                                used in MDA_Init MPI communications
C
C      Integer max_groups, max_procs, maxbuf, max_zones
C      Parameter(max_groups=512, max_procs=512,
C                 max_zones=max_groups)
C      Parameter (maxbuf=2*max_groups+max_procs+3)
C      Integer mdarun_group_id, mdarun_group_comm
C      Integer mdarun_group_leaders, mdarun_num_groups
C      Common /MDARUN/mdarun_group_id,
C                 MDARUN_GROUP_COMM,
C                 ;          mdarun_num_groups,
C                 ;          mdarun_group_leaders(0:max_groups)
C
C      MDA_Init Setup data - Must be input or set manually
C      See MDA_Init.f for definitions
C
C      Integer num_app_types,nprocs_per_group,
C                 num_app_groups, group_app_type,
C                 global_to_group_map
C      Common /MDADAT/num_app_types,num_app_groups,
C                 nprocs_per_group(0:max_procs),
C                 group_app_type(0:max_groups),
C                 global_to_group_map(0:max_procs)
C
C      Old MPIRUN Data Entities
C
C      Integer mpirun_app_id, mpirun_app_comm
C      Integer mpirun_app_leaders,mpirun_num_apps
C      Common /MPIRUN/mpirun_app_id,

```

```

;           MPIRUN_APP_COMM,
;
;           mpirun_num_apps,
;           mpirun_app_leaders(0:max_zones)
C

```

A.2 The FORTRAN 90 Modules - mdarun_data and mpirun_data

Module mdarun_data

```

Module mdarun_data

! Defines data from mda_init.f90 for
! multi-disciplinary runs

! where
!   mdarun_group_id - the group id for the current
!                     proc
!   MDARUN_GROUP_COMM - group communicator
!   mdarun_num_groups - number of groups defined
!   mdarun_group_leaders - global rank in MPI_COMM_WORLD
!                         of first process in group

! The following data is defined in MDA_Init and/or via input or
! hardwired code

!   global_to_group_map - parent group of each process in MPI_COMM_WORLD
!                         Defined in MDA_init. Dimension is
!                         (0:global_size-1)

! The following data MUST be input or hardwired into the user code
! that will start first (ie be proc 0 in MPI_COMM_WORLD)

!   num_app_types = number of different app types used
!                   defined via input or hardwired code
!   num_app_groups = number of different groups of processors
!   nprocs_per_group - number of processes in each group
!                      dimension should be (0:max_procs)
!                      defined via input or hardwired code
!   group_app_type - app type of each group (0:max_procs)
!                      dimension should be (0:max_procs)
!                      defined via input or hardwired code

! Implicit None

! Integer, Save :: mdarun_group_id
! Integer, Save :: MDARUN_GROUP_COMM
! Integer, Save :: mdarun_num_groups
! Integer, Save :: num_app_types
! Integer, Save :: num_app_groups

! The following two arrays are allocated in MDA_Init
!
```

```

    Integer, Save, Allocatable, Dimension(:) :: mdarun_group_leaders
    Integer, Save, Allocatable, Dimension(:) :: global_to_group_map

! NOTE

! The next two Allocatable arrays MUST be allocated by user before
! passing data to MDA_Init IF the user decides to hardcode the data
! and not input via file. Dimension should be (0:max_groups-1)

    Integer, Save, Allocatable, Dimension(:) :: nprocs_per_group
    Integer, Save, Allocatable, Dimension(:) :: group_app_type

! End Module mdarun_data

Module mpirun_data

    Module mpirun_data

! Defines data from mda_init.f90 for
! multi-disciplinary runs. Used to provide
! an interface to old MPIRUN common block data

! where
!     mpirun_app_id - the group id for the current
!                     proc
!     MPIRUN_APP_COMM - group communicator
!     mpiun_num_apps - number of groups defined
!     mpiun_app_leaders - global rank in MPI_COMM_WORLD
!                         of first process in group

! Implicit None

    Integer, Save :: mpirun_app_id
    Integer, Save :: MPIRUN_APP_COMM
    Integer, Save :: mpirun_num_apps

! NOTE

! Allocatable arrays MUST be allocated by user before passing data
! to MDA_Init IF the user decides to hardcode the data and not
! input via file

    Integer, Save, Allocatable, Dimension(:) :: mpirun_app_leaders

End Module mpirun_data

```

A.3 The C header file - mdarun.h

```

/* This file should be included AFTER the mpi.h file */
/* MDA/MPIRUN external data */

```

```

MPI_Comm MDARUN_GROUP_COMM;

```

```

int mdarun_group_id;
int mdarun_num_groups;
int *mdarun_group_leaders;
MPI_Comm MPIRUN_APP_COMM;
int MPIRUN_APP_ID;
int MPIRUN_NUM_APPS;
int *MPIRUN_APP_LEADERS;

/* MDA_Init setup data */
int num_app_types;
int num_app_groups;
int *nprocs_per_group;
int *group_app_type;
int *global_to_group_map;
/*
----- MDARUN Function Prototypes ---
*/
/* MDA_Init Function prototype */
void MDA_Init(int no_read, FILE *md_unit);

/* MDA_Intergroup_comm Function prototype */
void MDA_Intergroup_comm(int remote_group_number, int comm_tag,
                        MPI_Comm *group_to_group_comm);

/* MDA_Sameapp_comm Function prototype */
void MDA_Sameapp_comm(MPI_Comm *same_app_comm);

/* MDA_Sameapp_comm Function prototype */
void MDA_Mergeproc_comm(int *merge_list,int list_size, int merge_number,
                       MPI_Comm *merged_proc_comm);

/* MDA_Sameapp_comm Function prototype */
void MPI_Create_mergelist(int *group_list, int *local_rank_list,
                         int list_size, int *merge_list);

```

APPENDIX B. FORTRAN AND C VERSIONS OF Test_mda and Setup_mda

B.1 FORTRAN 90 Version of Test_mda

```
Program Test_mda
!
! Test program for mdarun routines
!
! Use mdarun_data module
!
    Use mdarun_data
    Use mpirun_data
!
! Implicit none to force explicit typeing
!
    Implicit None
!
! Include MPI Fortran header file
!
    Include 'mpif.h'
!
! Define Integer parameters
!
    Integer, Parameter :: max_buff=1000
!
! Temp fixed size arrays
!
    Real*8, Dimension(0:max_buff-1) :: send_buff
    Real*8, Dimension(0:max_buff-1) :: recv_buff
    Integer, Dimension(0:max_buff-1) :: int_send_buff
    Integer, Dimension(0:max_buff-1) :: int_recv_buff
    Integer, Dimension(MPI_STATUS_SIZE) :: status_mpi
!
! Temp allocatable array
!
    Integer, Allocatable, Dimension(:) :: INTERCOMM_Array
!
! Local Variables
!
    Integer :: SAMETYPE_COMM
    Integer :: MY_MERGED_COMM
    Integer :: global_size, my_global_rank
    Integer :: my_group_num, my_group_size, my_local_rank
    Integer :: my_app_type, nprocs, n, nn, np
    Integer :: send_err, recv_err, wait_err, ierr
    Integer :: send_request, recv_request, comm_tag_s, comm_tag_r
    Integer :: ou, ipass, same_size, same_rank
    Integer :: merge_size, merge_rank
!
```

```

! Start of by call MPI_Init
!
!     Call MPI_Init(ierr)
!
! Get global size and rank from MPI_COMM_WORLD
!
!     Call MPI_Comm_size(MPI_COMM_WORLD, global_size, ierr)
!     Call MPI_Comm_rank(MPI_COMM_WORLD, my_global_rank, ierr)
!
! Allocate Memory for INTERCOMM_Array
!
!     Allocate(INTERCOMM_Array(0:global_size-1))
!
! Define ou
!
!     ou = 9+my_global_rank
!
! Call Setup_mda which in turn calls MDA_Init, MDA_Sameapp_comm
! and MDA_Mergeproc_comm
!
!     Call Setup_mda(ou, &
!                   SAMETYPE_COMM, &
!                   MY_MERGED_COMM, &
!                   global_size, &
!                   INTERCOMM_Array)
!
! Initialize send/receive buffer arrays
!
!     Do n=0, max_buff-1
!         recv_buff(n) = 0.0
!         send_buff(n) = n
!         int_send_buff(n) = 0
!         int_recv_buff(n) = 0
!     EndDo
!
! Get mdarun group number, local rank, group size etc.
!
!     my_group_num = global_to_group_map(my_global_rank)
!     my_local_rank = my_global_rank - &
!                     mdarun_group_leaders(my_group_num)
!     my_group_size = nprocs_per_group(my_group_num)
!
! TEST OF MDARUN_GROUP_COMM COMMUNCATOR
!
! Using MDARUN_GROUP_COMM send/receive buffer to/from other
! members of the group
!
!     Write(ou, '(" TEST OF MDA_GROUP_COMM for group no. ', i2, &
! &           '' on local proc no. ', i2, '' with global rank = ', &
! &           i2)') my_group_num, my_local_rank, my_global_rank
!     Write(ou, '(" My group size = ', i2)') my_group_size

```

```

!
! SGI flush command for flushing I/O buffer for ou at this point
! Use flush_ for IBM SP

!
call flush(ou)

!
! Loop through group size and send/recieve with non-blocking routine
! MPI_Isend and blocking MPI_Recv

!
Do n=0, my_group_size-1
  If (my_local_rank.ne.n) Then
    comm_tag_s = 1000 + my_local_rank
    Call MPI_Isend(send_buff(0), &
                  max_buff, &
                  MPI_DOUBLE_PRECISION, &
                  n, &
                  comm_tag_s, &
                  MDARUN_GROUP_COMM, &
                  send_request, &
                  send_err)
    Call MPI_Wait(send_request, status_mpi, wait_err)
  EndIf
EndDo
Do n=0, my_group_size-1
  If (my_local_rank.ne.n) Then
    comm_tag_r = 1000+n
    Call MPI_Recv(recv_buff(0), &
                  max_buff, &
                  MPI_DOUBLE_PRECISION, &
                  n, &
                  comm_tag_r, &
                  MDARUN_GROUP_COMM, &
                  status_mpi, &
                  recv_err)
  EndIf
EndDo
!
! Check Message
!
ipass = 1
Do nn=0,max_buff-1
  If (recv_buff(nn).ne.send_buff(nn)) ipass=0
EndDo
!
If (ipass.eq.1) Then
  Write(ou, '(" MDARUN_GROUP_COMM Test ", &
  &           " message from proc. ", &
  &           i2,'' passed on proc with global rank = '', &
  &           i2)') n, my_global_rank
  call flush(ou)
  Else
    Write(ou, '(" MDARUN_GROUP_COMM Test ", &
  &           " message from proc. ", &

```

```

&      i2, '' failed on proc with global rank = '', &
&      i2)' n, my_global_rank
call flush(ou)
EndIf
Do nn=0,max_buff-1
recv_buff(nn) = 0.0
EndDo
EndIf
EndDo

!
! TEST OF SAMEAPP COMMUNICATOR
!
my_app_type = group_app_type(my_group_num)

Write(ou, '(''0 TEST OF SAMETYPE_COMM COMMUNICATOR '')')
Write(ou, '('' I am a proc. with app type = '', i2)'') &
my_app_type
call flush(ou)

Call MPI_Comm_size(SAMETYPE_COMM, same_size, ierr)
Call MPI_Comm_rank(SAMETYPE_COMM, same_rank, ierr)

!
Write(ou, '('' Same_size = '', i2, '' on proc no. '', &
&           i2)'') same_size, my_global_rank
Write(ou, '('' Same_rank = '', i2, '' on proc no. '', &
&           i2)'') same_rank, my_global_rank
call flush(ou)

!
! Use MPI_Allreduce to send array with just the global rank
! in the local rank number array location. The returned array
! should have the list of local ranks for each SAMEAPP_COMM
! member
!
int_send_buff(same_rank) = my_global_rank
Call MPI_Allreduce(int_send_buff(0), &
int_recv_buff(0), &
same_size, &
MPI_INTEGER, &
MPI_SUM, &
SAMETYPE_COMM, &
ierr)

!
Write(ou, '('' Global rank list for proc no. '', i2, &
&           '' SAMETYPE_COMM '')') my_global_rank
call flush(ou)
Do n=0,same_size-1
Write(ou, '('' Local rank in SAMETYPE_COMM= '', i2, &
&           '' Global rank ='', i2)'') &
n, int_recv_buff(n)
call flush(ou)

```

```

EndDo

! TEST OF INTERCOMMUNICATORS

! Group leaders send/recieve to/from all procs in the
! other groups. All procs will receive/send from/to group leaders

    Write(ou, '( '' TEST OF INTERCOMMUNICATORS '' )')
    Write(ou, '( '' Test of Group to Group Intercomm, '' , &
    &           '' on local proc no. '' , i2, '' with global rank = '' , &
    &           i2)' ) my_local_rank, my_global_rank
    call flush(ou)

! Reinitialize send/receive buffers

Do n=0, max_buff-1
    recv_buff(n) = 0.0
    send_buff(n) = n+1
    int_send_buff(n) = 0
    int_recv_buff(n) = 0
EndDo

! Loop through all groups and send/recieve with non-blocking routine
! MPI_Isend and blocking MPI_Recv

! Send to other groups procs if I am a group leader

    comm_tag_s = 3000
    If (my_local_rank.eq.0) Then
        Do n=0, mdarun_num_groups-1
            If (my_group_num.ne. n) Then
                nprocs = nprocs_per_group(n)
                Do np=0, nprocs-1
                    Call MPI_Isend(send_buff(0), &
                        max_buff, &
                        MPI_DOUBLE_PRECISION, &
                        np, &
                        comm_tag_s, &
                        INTERCOMM_Array(n), &
                        send_request, &
                        send_err)
                    Call MPI_Wait(send_request, status_mpi, wait_err)
                EndDo
            EndIf
        EndDo
    EndIf
    comm_tag_r = 3000
    Do n=0, mdarun_num_groups-1
        If (my_group_num.ne.n) Then
            Call MPI_recv(recv_buff(0), &
                max_buff, &

```

```

        MPI_DOUBLE_PRECISION, &
        0, &
        comm_tag_r, &
        INTERCOMM_Array(n), &
        status_mpi, &
        recv_err)

!
! Check Message
!

    ipass = 1
    Do nn=0,max_buff-1
        If(recv_buff(nn).ne.send_buff(nn)) ipass=0
    EndDo

    !
    If (ipass.eq.1) Then
        Write(ou, '(,'' INTERCOMM TEST '',/, &
        &      '' message from group leader of group no. '', &
        &      i2, &
        &      '' passed on proc no = '',i2/, &
        &      '' who is a member of group no. '', &
        &      i2)) n, my_global_rank, my_group_num
        call flush(ou)
    Else
        Write(ou, '(,'' INTERCOMM TEST '',/, &
        &      '' message from group leader of group no. '', &
        &      i2, &
        &      '' failed on proc no = '', i2/, &
        &      '' who is a member of group no. '', &
        &      i2)) n, my_global_rank, my_group_num
        call flush(ou)
    EndIf
    Do nn=0,max_buff-1
        recv_buff(nn) = 0.0
    EndDo
    EndIf
EndDo

!
! TEST OF MY_MERGED_COMM COMMUNICATOR
!

    Write(ou, '(,''0 TEST OF MY_MERGED_COMM COMMUNICATOR '''))
    Write(ou, '(,'' I am a proc. with app type = '', i2)'') &
        my_app_type
    call flush(ou)

    !
    Call MPI_Comm_size(MY_MERGED_COMM, merge_size, ierr)
    Call MPI_Comm_rank(MY_MERGED_COMM, merge_rank, ierr)

    !
    Write(ou, '(,'' Merge_size = '', i2,'' on proc no. '', &
    &              i2)) merge_size, my_global_rank
    Write(ou, '(,'' Merge_rank = '', i2,'' on proc no. '', &

```

```

&           i2)') merge_rank, my_global_rank

call flush(ou)

! Use MPI_Allreduce to send array with just the global rank
! in the local rank number array location. The returned array
! should have the list of global ranks for each MY_MERGED_COMM
! member

! Reinitialize int send/receive buffs

Do nn=0,max_buff-1
    int_send_buff(nn) = 0
    int_recv_buff(nn) = 0
EndDo

int_send_buff(merge_rank) = my_global_rank
Call MPI_Allreduce(int_send_buff(0), &
                  int_recv_buff(0), &
                  merge_size, &
                  MPI_INTEGER, &
                  MPI_SUM, &
                  MY_MERGED_COMM, &
                  ierr)

! Write(ou, '( ' Global rank list on proc no. ', i2, &
! &           ' for MY_MERGED_COMM ')') my_global_rank
call flush(ou)
Do n=0,merge_size-1
    Write(ou, '( ' Local rank in MY_MERGED_COMM= ', i2, &
    &           ' Global rank =', i2)') &
    n, int_recv_buff(n)
call flush(ou)
EndDo

! Use MPI_Allreduce to send array with just the app type
! in the local rank number array location. The returned array
! should have the list of app_types for each MY_MERGED_COMM
! member

! Reinitialize int send/receive buffs

Do nn=0,max_buff-1
    int_send_buff(nn) = 0
    int_recv_buff(nn) = 0
EndDo

int_send_buff(merge_rank) = my_app_type
Call MPI_Allreduce(int_send_buff(0), &
                  int_recv_buff(0), &
                  merge_size, &

```

```

        MPI_INTEGER,  &
        MPI_SUM,  &
        MY_MERGED_COMM,  &
        ierr)

!
Write(ou, '(" App_Type list on proc no. ', i2, '&
&           '' for MY_MERGED_COMM '')') my_global_rank
call flush(ou)
Do n=0,merge_size-1
    Write(ou, '(" Local rank in MY_MERGED_COMM= ', i2, '&
&           '' App_type =', i2)') &
        n, int_recv_buff(n)
call flush(ou)
EndDo

!
! Write Exit message and quit
!
Write(*,(' Fortran 90 proc no. ', i2, '' says Adios '')) &
my_global_rank
!
Call MPI_Finalize(ierr)

!
Stop
!
End

```

B.2 FORTRAN 90 Version of Setup_mda

```
Subroutine Setup_mda(ou, &
                     SAMETYPE_COMM,&
                     MY_MERGED_COMM, &
                     Int_size, &
                     INTERCOMM_Array)

! Test setup routine for MDA run routines

Use mdarun_data
Use mpirun_data

Implicit None

Include 'mpif.h'

Arg list variables

Integer, Intent(IN) :: int_size
Integer, Intent(INOUT) :: SAMETYPE_COMM
Integer, Intent(INOUT) :: MY_MERGED_COMM
Integer, Intent(INOUT), Dimension(0:int_size-1) :: INTERCOMM_Array

Local Variables

Integer, Parameter :: md_unit=7
Integer :: n, nn, ierr, merge_num
Integer :: global_size
Integer :: my_global_rank
Integer :: no_read
Integer :: ou
Integer :: num_merges
Integer :: my_group_num
Integer :: my_local_rank
Integer :: my_app_type
Integer :: comm_tag
Integer :: tmp_comm

Integer, Allocatable, Dimension(:) :: group_list
Integer, Allocatable, Dimension(:) :: local_rank_list
Integer, Allocatable, Dimension(:) :: merge_list

Call MPI_Comm_size(MPI_COMM_WORLD, global_size, ierr)
Call MPI_Comm_rank(MPI_COMM_WORLD, my_global_rank, ierr)

Allocate(group_list(0:global_size-1))
Allocate(local_rank_list(0:global_size-1))
Allocate(merge_list(0:global_size-1))
```

```

! Call MDA_Init to Initialize MDARUN data and
! group communicator

    no_read=0
    Call MDA_Init(no_read,md_unit)
    num_merges = nprocs_per_group(0)

! Get my group number, and compute my_local_rank
! using mdarun data

    my_group_num = global_to_group_map(my_global_rank)
    my_local_rank = my_global_rank - &
                    mdarun_group_leaders(my_group_num)
    my_app_type = group_app_type(my_group_num)

    Write(ou,'('' MY SETUP DATA - PROC = '',i2)') &
    my_global_rank
    Write(ou,'('' my_group_num = '',i2)') my_group_num
    Write(ou,'('' my_local_rank = '',i2)') my_local_rank
    Write(ou,'('' my_app_type = '',i2)') my_app_type

! ***** TEST OF MDA_SAMEAPP_COMM *****
! Call MDA_Sameapp_comm to create a communicator
! With the same app type as me

    Call MDA_Sameapp_comm(SAMETYPE_COMM)

! ***** TEST OF MDA_MERGEPROC_COMM/MDA_MERGELIST_COMM *****
! Form a merged communicator that consists of the processes
! with the same local rank in each group. This example
! assumes all the groups contain the same number of
! processess. For compatability with the C code the
! output is saved as temp varaible which is used to
! create MY_MERGED_COMM by a call to MPI_Comm_dup

    Do n=0, num_merges-1
        merge_num=n
        Do nn=0, mdarun_num_groups-1
            group_list(nn) = nn
            local_rank_list(nn)=n
        EndDo
        Call MDA_Create_mergelist(group_list,  &
                                    local_rank_list,  &
                                    mdarun_num_groups,  &
                                    merge_list(0))
        Call MDA_Mergeproc_comm(merge_list(0),   &
                               mdarun_num_groups,  &
                               merge_num,   &
                               tmp_comm)

```

```

    If (my_local_rank .eq. n) then
        Call MPI_Comm_dup(tmp_comm,MY_MERGED_COMM,ierr)
    EndIf
EndDo

! **** TEST OF MDA_INTEGROUP_COMM *****
!
! Create an array containing intergroup communicators
! to other group leaders
!

Do n=0,mdarun_num_groups-1
    comm_tag = 1000
    Call MDA_Intergroup_Comm(n,comm_tag, &
                           INTERCOMM_Array(n))
EndDo

!
Write(ou,'( COMMUNICATOR DATA FOR GLOBAL PROC NO. ', &
&           i2') my_global_rank
Write(ou,'( My group communicator MDARUN_GROUP_COMM = ', &
&           i2') MDARUN_GROUP_COMM
Write(ou,'( My Same App communicator = ', &
&           i2') SAMETYPE_COMM
Write(ou,'( My Merged App communicator = ', &
&           i2') MY_MERGED_COMM
Write(ou,'( My INTERCOMM_ARRAY LIST '''))
Do n=0,mdarun_num_groups-1
    Write(ou,'( Group no. = ', i2, ' Intercomm = ', &
&           i2') n, INTERCOMM_ARRAY(n)
Enddo
call flush(ou)

!
End Subroutine Setup_mda

```

B.3 C Version of Test_mda

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include "mdarun.h"
#define max_buff 1000

/*
!
! Temp allocatable array
!
!

! Local Variables
!
*/
MPI_Comm SAMETYPE_COMM;
MPI_Comm MY_MERGED_COMM;
MPI_Request send_request, recv_request;
MPI_Comm *INTERCOMM_Array;

int global_size, my_global_rank;
int my_group_num, my_group_size, my_local_rank;
int my_app_type, nprocs, n, nn, np;
int comm_tag_s, comm_tag_r;
int ipass, same_size, same_rank;
int my_unit, merge_size, merge_rank;
char filename[8];
char fname1[2];
FILE *ou;

/*
!
! Temp fixed size arrays
!
*/

double send_buff[max_buff];
double recv_buff[max_buff];
int int_send_buff[max_buff];
int int_recv_buff[max_buff];
MPI_Status status_mpi;

void Setup_mda(FILE *ou, MPI_Comm *SAMETYPE_COMM,
               MPI_Comm *MY_MERGED_COMM,
               int int_size,
               MPI_Comm *INTERCOMM_Array);

int main(argc, argv)
int argc;
char **argv;
```

```

{
/*
!
!     Test program for mdarun routines
!
!

*/
/* The following externals are replaced by
   the type defs in mdarun.h. Not needed here
   but I'll leave them so the user knows what's
   external without having to look at mdarun.h

extern MPI_Comm MDARUN_GROUP_COMM;
extern int mdarun_group_id;
extern int mdarun_num_groups;
extern int *mdarun_group_leaders;
extern MPI_Comm MPIRUN_APP_COMM;
extern int MPIRUN_APP_ID;
extern int MPIRUN_NUM_APPS;
extern int *MPIRUN_APP_LEADERS;
extern int num_app_types;
extern int num_app_groups;
extern int *nprocs_per_group;
extern int *group_app_type;
extern int *global_to_group_map;
*/
/*
!
! Start of by call MPI_Init
!
*/
        MPI_Init(&argc,&argv);

/*
!
! Get global size and rank from MPI_COMM_WORLD
!
*/
        MPI_Comm_size(MPI_COMM_WORLD, &global_size);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_global_rank);
/*
!
! Allocate Memory for INTERCOMM_Array
!
*/
        INTERCOMM_Array=(MPI_Comm *)malloc(global_size*sizeof(MPI_Comm));
/*
!
! Define ou
!
*/

```

```

    my_unit = 9+my_global_rank;
    sprintf(fname1, "%i",my_unit);
    strcpy(filename,"cout.");
    strcat(filename,fname1);
    if ((ou=fopen(filename,"w")) == NULL)
    {
        fprintf(stdout," File %s not opened\n",filename);
        MPI_Finalize();
        exit(1);
    }
/*
!
! Setup_mda which in turn calls MDA_Init, MDA_Sameapp_comm
! and MDA_Mergeproc_comm
!
*/
Setup_mda(ou,
           &SAMETYPE_COMM,
           &MY_MERGED_COMM,
           global_size,
           INTERCOMM_Array);
/*
!
! Initialize send/receive buffer arrays
!
*/
for(n=0;n<=(max_buff-1);n++)
{
    recv_buff[n] = 0.0;
    send_buff[n] = n;
    int_send_buff[n] = 0;
    int_recv_buff[n] = 0;
}
/*
!
! Get mdarun group number, local rank, group size etc.
!
*/
my_group_num = global_to_group_map[my_global_rank];
my_local_rank = my_global_rank -
                mdarun_group_leaders[my_group_num];
my_group_size = nprocs_per_group[my_group_num];
/*
!
! TEST OF MDARUN_GROUP_COMM COMMUNCATOR
!
! Using MDARUN_GROUP_COMM send/receive buffer to/from other
! members of the group
!
*/
fprintf(ou, " TEST OF MDA_GROUP_COMM for group no. %i\n",my_group_num);

```

```

fprintf(ou, " on local proc no. %i with global rank = %i\n",
        my_local_rank, my_global_rank);
fprintf(ou, "My group size = %i\n", my_group_size);

/*
!
! Loop through group size and send/recieve with non-blocking routine
! MPI_Isend and the blocking MPI_Recv
!
*/
for(n=0;n<=(my_group_size-1);n++)
{
    if (my_local_rank != n)
    {
        comm_tag_s = 1000 + my_local_rank,
        MPI_Isend(send_buff,
                  max_buff,
                  MPI_DOUBLE,
                  n,
                  comm_tag_s,
                  MDARUN_GROUP_COMM,
                  &send_request);
        MPI_Wait(&send_request, &status_mpi);
    }
}
for(n=0;n<=(my_group_size-1);n++)
{
    if (my_local_rank != n)
    {
        comm_tag_r = 1000+n;
        MPI_Recv(recv_buff,
                  max_buff,
                  MPI_DOUBLE,
                  n,
                  comm_tag_r,
                  MDARUN_GROUP_COMM,
                  &status_mpi);
    }
}

/*
!
! Check Message
!
*/
ipass = 1;
for(nn=0;nn<=(max_buff-1);nn++)
    if(recv_buff[nn] != send_buff[nn]) ipass=0;
if (ipass == 1)
{
    fprintf(ou, "MDARUN_GROUP_COMM Test \n");
    fprintf(ou, "message from proc. %i\n",n);
    fprintf(ou," passed on proc with global rank = %i\n",
            my_global_rank);
}

```

```

        }
    else
    {
        fprintf(ou, "MDARUN_GROUP_COMM Test \n");
        fprintf(ou, "message from proc. %i\n",n);
        fprintf(ou," failed on proc with global rank = %i\n",
                my_global_rank);
    }
    for(nn=0;nn<=(max_buff-1);nn++)
        recv_buff[nn] = 0.0;
}
}

/*
!
! TEST OF SAMEAPP COMMUNICATOR
!
*/
my_app_type = group_app_type[my_group_num];

fprintf(ou, "\nTEST OF SAMETYPE_COMM COMMUNICATOR \n");
fprintf(ou, "I am a proc. with app type = %i\n",
        my_app_type);

MPI_Comm_size(SAMETYPE_COMM, &same_size);
MPI_Comm_rank(SAMETYPE_COMM, &same_rank);

fprintf(ou, " Same_size = %i on proc no. %i\n",
        same_size, my_global_rank);
fprintf(ou, " Same_rank = %i on proc no. %i\n",
        same_rank, my_global_rank);

/*
!
! Use MPI_Allreduce to send array with just the global rank
! in the local rank number array location. The returned array
! should have the list of global ranks for each SAMEAPP_COMM
! member
!
*/
int_send_buff[same_rank] = my_global_rank;
MPI_Allreduce(int_send_buff,
              int_recv_buff,
              same_size,
              MPI_INT,
              MPI_SUM,
              SAMETYPE_COMM);

fprintf(ou, "Global rank list for proc no. %i for SAMETYPE_COMM\n",
        my_global_rank);

```

```

for(n=0;n<=(same_size-1);n++)
    fprintf(ou, "Local rank in SAMETYPE_COMM= %i Global rank = %i\n",
            n, int_recv_buff[n]);

/*
!
! TEST OF INTERCOMMUNICATORS
!
! Group leaders send/recieve to/from all procs in the
! other groups. All procs will receive/send from/to group leaders
!
*/
fprintf(ou, "TEST OF INTERCOMMUNICATORS \n");
fprintf(ou, " Test of Group to Group Intercomm");
fprintf(ou, "on local proc no. %i with global rank = %i\n",
        my_local_rank, my_global_rank);

/*
!
! Reinitialize send/receive buffers
!
*/
for(n=0; n<=(max_buff-1);n++)
{
    recv_buff[n] = 0.0;
    send_buff[n] = n+1;
    int_send_buff[n] = 0;
    int_recv_buff[n] = 0;
}
/*
!
! Loop through all groups and send/recieve with non-blocking routine
! MPI_Isend and blocking MPI_Recv
!
! Send to other groups procs if I am a group leader
!
*/
comm_tag_s = 3000;
if (my_local_rank == 0)
{
    for(n=0;n<=(mdarun_num_groups-1);n++)
    {
        if (my_group_num != n)
        {
            nprocs = nprocs_per_group[n];
            for(np=0;np<=(nprocs-1);np++)
                MPI_Isend(send_buff,
                           max_buff,
                           MPI_DOUBLE,
                           np,

```

```

        comm_tag_s,
        INTERCOMM_Array[n],
        &send_request);
    MPI_Wait(&send_request, &status_mpi);
}
}
}
}
comm_tag_r = 3000;
for(n=0;n<=(mdarun_num_groups-1);n++)
{
    if (my_group_num != n)
    {
        MPI_Recv(recv_buff,
                  max_buff,
                  MPI_DOUBLE,
                  0,
                  comm_tag_r,
                  INTERCOMM_Array[n],
                  &status_mpi);
    }
    /*
    !
    ! Check Message
    !
    */
    ipass = 1;
    for(nn=0;nn<=(max_buff-1);nn++)
        if(recv_buff[nn] != send_buff[nn]) ipass=0;

    if (ipass == 1)
    {
        fprintf(ou, "INTERCOMM TEST \n");
        fprintf(ou, "message from group leader of group no. %i\n",
                n);
        fprintf(ou, " passed on proc no = %i\n",
                my_global_rank);
        fprintf(ou," who is a member of group no. %i\n",
                my_group_num);

    }
    else
    {
        fprintf(ou, "INTERCOMM TEST \n");
        fprintf(ou, "message from group leader of group no. %i\n",
                n);
        fprintf(ou, " failed on proc no = %i\n",
                my_global_rank);
        fprintf(ou," who is a member of group no. %i\n",
                my_group_num);

    }
    for(nn=0;nn<=(max_buff-1);nn++)

```

```

        recv_buff[nn] = 0.0;
    }
}
/*
!
! TEST OF MY_MERGED_COMM COMMUNICATOR
!
!
*/
fprintf(ou, "\nTEST OF MY_MERGED_COMM COMMUNICATOR\n");
fprintf(ou, "I am a proc. with app type = %i\n",
        my_app_type);

MPI_Comm_size(MY_MERGED_COMM, &merge_size);
MPI_Comm_rank(MY_MERGED_COMM, &merge_rank);

fprintf(ou, " Merge_size = %i on proc no. %i\n",
        merge_size, my_global_rank);
fprintf(ou, " merge_rank = %i on proc no. %i\n",
        merge_rank, my_global_rank);

/*
!
! Use MPI_Allreduce to send array with just the global rank
! in the local rank number array location. The returned array
! should have the list of global ranks for each MY_MERGED_COMM
! member
!
! Reinitialize int send/receive buffs
!
*/
for(nn=0;nn<=(max_buff-1);nn++)
{
    int_send_buff[nn] = 0;
    int_recv_buff[nn] = 0;
}

int_send_buff[merge_rank] = my_global_rank;
MPI_Allreduce(int_send_buff,
              int_recv_buff,
              merge_size,
              MPI_INT,
              MPI_SUM,
              MY_MERGED_COMM );

fprintf(ou, "Global rank list on proc no. %i for MY_MERGED_COMM\n",
        my_global_rank);

for(n=0;n<=(merge_size-1);n++)

```

```

        fprintf(ou, "Local rank in MY_MERGED_COMM= %i Global rank = %i\n",
                n, int_recv_buff[n]);
/*
!
! Use MPI_Allreduce to send array with just the app type
! in the local rank number array location. The returned array
! should have the list of app_types for each MY_MERGED_COMM
! member
!
! Reinitialize int send/receive buffs
!
*/
for(nn=0;nn<=(max_buff-1);nn++)
{
    int_send_buff[nn] = 0;
    int_recv_buff[nn] = 0;
}

int_send_buff[merge_rank] = my_app_type;
MPI_Allreduce(int_send_buff,
              int_recv_buff,
              merge_size,
              MPI_INT,
              MPI_SUM,
              MY_MERGED_COMM);

fprintf(ou, "App_Type list on proc no. %i for MY_MERGED_COMM\n",
        my_global_rank);

for(n=0;n<=(merge_size-1);n++)
    fprintf(ou, "Local rank in MY_MERGED_COMM= %i App Type = %i\n",
            n, int_recv_buff[n]);

/*
!
! fprint Exit message and quit
!
*/
fprintf(stdout, "C proc no. %i says Adios\n",
        my_global_rank);

MPI_Finalize();
}

```

B.4 C Version of Setup_mda

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

/* Make MPI/MDA data external */
extern MPI_Comm MDARUN_GROUP_COMM;
extern int mdarun_group_id;
extern int mdarun_num_groups;
extern int *mdarun_group_leaders;
/* MDA_Init setup data */
extern int *nprocs_per_group;
extern int *group_app_type;
extern int *global_to_group_map;

void MDA_Init(int no_read, FILE *md_unit);
void MDA_Intergroup_comm(int remote_group_number, int comm_tag,
                        MPI_Comm *group_to_group_comm);
void MDA_Sameapp_comm(MPI_Comm *same_app_comm);
void MPI_Create_mergelist(int *group_list, int *local_rank_list,
                         int list_size, int *merge_list);
void MDA_Mergeproc_comm(int *merge_list, int list_size, int merge_number,
                       MPI_Comm *merged_proc_comm);

void Setup_mda(FILE *ou,
               MPI_Comm *SAMETYPE_COMM,
               MPI_Comm *MY_MERGED_COMM,
               int int_size,
               MPI_Comm *INTERCOMM_Array)
{

/*
! Setup function for test of mdarun
! C version
!
!
!
!     Arg list variables
!
!
!
!     Local Variables
!
*/
FILE *md_unit;
int n, nn, merge_num;
int global_size;
int my_global_rank;
int no_read;
int num_merges;
int my_group_num;
```

```

int  my_local_rank;
int  my_app_type;
int  comm_tag;
int *group_list;
int *local_rank_list;
int *merge_list;
MPI_Comm tmp_comm;
/*
! Call MPI_Comm size and rank
*/
    MPI_Comm_size(MPI_COMM_WORLD, &global_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_global_rank);
/*
! Get temp array memory
*/
    group_list = (int *)malloc(global_size*sizeof(int));
    local_rank_list = (int *)malloc(global_size*sizeof(int));
    merge_list = (int *)malloc(global_size*sizeof(int));

/*
!     Call MDA_Init to Initialize MDARUN data and
!     group communicator
!
*/
no_read=0;
md_unit = fopen("mdainit.dat","r");
MDA_Init(no_read,md_unit);
num_merges = nprocs_per_group[0];
/*    merge_comm_array=(MPI_Comm *)malloc(global_size*sizeof(MPI_Comm *));  */
/*
!
!     Get my group number, and compute my_local_rank
!     using mdarun data
!
*/
my_group_num = global_to_group_map[my_global_rank];
my_local_rank = my_global_rank -
    mdarun_group_leaders[my_group_num];
my_app_type = group_app_type[my_group_num];
fprintf(ou,"MY SETUP DATA - PROC = %i\n",my_global_rank);
fprintf(ou,"my_group_num = %i\n", my_group_num);
fprintf(ou," my_local_rank = %i\n", my_local_rank);
fprintf(ou," my_app_type = %i\n", my_app_type);
/*
!
! ***** TEST OF MDA_SAMEAPP_COMM *****
!
!     Call MDA_Sameapp_comm to create a communicator
!     With the same app type as me
!
*/

```

```

MDA_Sameapp_comm(SAMETYPE_COMM) ;
/*
!
! **** TEST OF MDA_MERGEPROC_COMM/MDA_MERGELIST_COMM *****
!
! Form a merged communicator that consists of the processes
! with the same local rank in each group. This example
! assumes all the groups contain the same number of
! processess. For test purposes the output from the
! call to MDA_Mergeproc_comm is saved in an array.
!
*/
for(n=0;n<=(num_merges-1);n++)
{
    merge_num=n;
    for(nn=0;nn<=(mdarun_num_groups-1);nn++)
    {
        group_list[nn] = nn;
        local_rank_list[nn]=n;
    }
    MDA_Create_mergelist(group_list,
                          local_rank_list,
                          mdarun_num_groups,
                          merge_list);
    MDA_Mergeproc_comm(merge_list,
                       mdarun_num_groups,
                       merge_num,
                       &tmp_comm);
    if (my_local_rank == n)
        MPI_Comm_dup(tmp_comm, MY_MERGED_COMM);

}
/*
!
! ***** TEST OF MDA_INTEGROUP_COMM *****
!
! Create an array containing intergroup communicators
! to other group leaders
!
*/
for(n=0;n<=(mdarun_num_groups-1);n++)
{
    comm_tag = 1000;
    MDA_Intergroup_comm(n,comm_tag,
                        &INTERCOMM_Array[n]);
}
fprintf(ou,"COMMUNICATOR DATA FOR GLOBAL PROC NO. %i\n",
        my_global_rank);
fprintf(ou,"My group communicator MDARUN_GROUP_COMM = %i\n",
        (int)MDARUN_GROUP_COMM);
fprintf(ou,"My Same App communicator = %i\n",

```

```
    (int)*SAMETYPE_COMM);  
    fprintf(ou,"My Merged App communicator = %i\n",  
           (int)*MY_MERGED_COMM);  
    fprintf(ou,"My INTERCOMM_ARRAY LIST \n");  
    for(n=0;n<=(mdarun_num_groups-1);n++)  
        fprintf(ou," Group no. = %i Intercomm = %i\n",  
                n, (MPI_Comm)INTERCOMM_Array[n]);  
    fflush(ou);  
}
```